Introduction to Programming

Block Tutorial C/C++

Michael Bader

Master's Program
"Computational Science and Engineering"

C/C++ Tutorial – Overview

- From Maple to C
- Variables, Operators, Statements
- Functions: declaration, definition, parameters
- Arrays and Pointers
- From Pointers to Dynamical Data Structures
- Object Oriented Programming: From C to C++

Thursday Lesson – Overview

- Pointers
- "Call by reference"
- Introduction to Pointer arithmetic
- Structs
- Dynamical data structures

Pointers – The * and & Operators

- the * operator "dereferences" a pointer, i.e. turns the pointer into the corresponding variable
- the & operator computes the address of a variable

Example:

```
int a=4,b=5;
int* pa;
pa = &a; *pa = 7;
pa = &b; *pa = 8;
printf("a=%d, b=%d\n",a,b);
```

"Call by Reference"

When using arrays as parameters, we can permanently modify array elements inside a function.

$\Rightarrow$ we can use this feature deliberately using pointers

Example:

```
void increase(int* a) { *a = *a + 1; }
main() {
   int a=5;
   increase(&a); printf("%d\n",a);
}
```

This technique is called **"Call by Reference"**.

scanf() revisited

We've been using "call by reference" every time we called the `scanf` function:

```
int main() {
   int a;
   scanf("%d", &a );
}
```

Remember: "call by reference" requires

- passing **addresses** instead of **values** to parameters
- dereferencing the addresses inside the function at every access

$\Rightarrow$ the value of the referenced variable can be changed permanently!

Simple Pointer Arithmetics

C allows us to do certain calculations with pointers.

Example: compute the length of a string

```
int strlength(const char* string) {
   int i=0;
   while( *string != '\0' ) {
      string++; i++;
   }
   return i;
}
```

Pointer Arithmetics and Arrays

Example: compute the length of a unicode string (16bit-characters)

```
int strlength(const short* string) {
   int i=0;
   while( *string != 0 ) { string++; i++; }
   return i;
}
```

Apparently, the ++-operator is "type sensitive".

Pointer Arithmetics and Arrays (2)

Example: accessing array elements

```
int main() {
   int i;
   int primes[] = {2,3,5,7,11,13,17,19};
   for(i=0;i<8;i++)
      printf("%d\n", *(primes+i) );
}
```

- `primes+i` refers to the address `primes+i*sizeof(int)`;
- `x[i]` is actually a short-cut for `*(x+i)`.

Pointer Arithmetics – Overview

The following pointer operations are allowed:

- the value of a pointer variable or `NULL` may be assigned to a pointer (`NULL` is an "invalid" address defined in `<stdio.h>`);
- Pointer variables may be compared using the usual operators (==, !=, <, >, etc.);
- The comparison `pnt==NULL` is allowed.

The following pointer operations are "type-sensitive":

- computing the "difference" between two pointers, i.e. `pnt1-pnt2` yields the number of elements between `pnt1` and `pnt2`;
- adding an integer to a pointer (see example).
- increasing or decreasing a pointer (using ++ or --, see example).

Structs – Types for Structured Data

Syntax:

```
struct <name> {
   <type> <name>;
   ...
} ;
```

Purpose:

- group variables
- design more complicated data structures

Structs – Example

```
struct date { int day;
              char* month;
              int year; };
struct date birthday, anniversary;
```

Structs may be "anonymous" or nested:

```
struct { struct date start;
         struct date end;
         char destination[100]; } holiday;
```

Note, that `holiday` is a variable; a `struct holiday` does not exist!

Structs – Accessing Members

Struct members can be accessed using the .-operator:

```
birthday.day = 13;
birthday.month = (char*) malloc(4*sizeof(char));
birthday.year = 1977;

holiday.start.year = holiday.end.year = 2001;
holiday.start.day = 11;
strcpy(holiday.destination,"Mallorca");
```

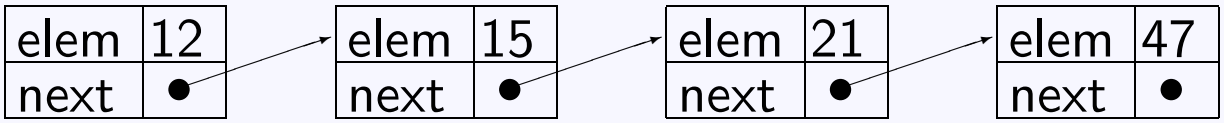
Self Referencing Pointers in Structs

C does not allow a struct to be a member of itself.
However, you **can** include a pointer to the struct!

Example:

```
struct list {
    int elem;
    struct list* next;
};
```

Such a data structure is called a **linked list**:



Terminating Lists – the `NULL` Pointer

The last element of a list would be one that has no successor, i.e. its member `next` does not point to a valid list element.

In practice, it is difficult to check whether a pointer points to a valid address.

⇒ C provides a dedicated value for "invalid" pointers: `NULL`.

`NULL` is defined in library `<stdio.h>`.
It will never represent a valid address.

⇒ always set pointers to `NULL` unless they do reference a "valid" address.

⇒ the last element of a list should "point" to `NULL`.

Giving Types a new Name – `typedef`

Sometimes, the names of `struct` types can be difficult to read.

C provides the `typedef` command to make things easier and more readable.

Syntax: `typedef <type> <name>;`

Example:

```
struct list {
   int elem;
   struct list *next;
};
typedef struct list* List;
List a,b;
```

Attaching an Element to a List

Recursive function:, called f.e. like `l = attach(l,5)`

```
List attach(List l, int value) {
   if ( l == NULL ) {
      l = (List) malloc(sizeof(struct list));
      l->elem = value;
      l->next = NULL;   /* l-> is equiv. to (*l). */
      return l;
   } else {
      l->next = attach(l->next, value) ;
      return l;
   }
}
```

This approach avoids using a call-by-reference parameter for `l`!

Attaching an Element to a List (2)

Iterative program:

```
List attach(List l, int value) {
   if ( l == NULL ) {
      l = (List) malloc(sizeof(struct list));
      l->elem = value; l->next = NULL;
      return l;
   } else {
      while (l->next != NULL) l = l->next;
      l->next = (List) malloc(sizeof(struct list));
      l->next->elem = value;
      l->next->next = NULL;
   } ;
   return l;
}
```

Deleting an Element from a List

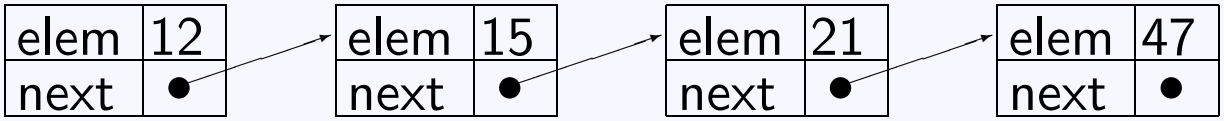
Recursive program:

```
List delete(List l, int value) {
   if ( l == NULL ) return NULL;
   l->next = delete(l->next,value);
   if (l->elem == value)
        return l->next;
   else return l;
}
```

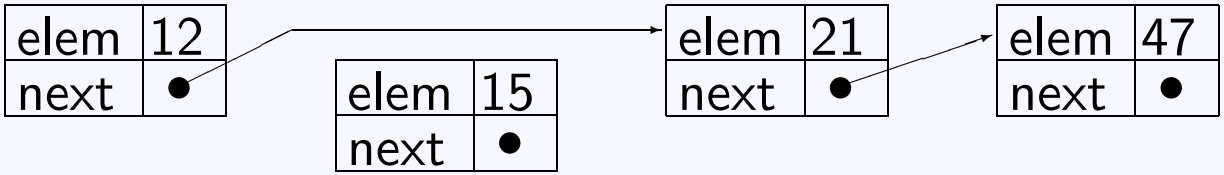
Question: What happens to "deleted" elements?

Deleting an Element from a List (2)

Before deletion:



After deletion:



- deleted element can no longer be accessed
- but still occupies allocated memory

$\Rightarrow$ technique is needed to delete list elements

free()

Syntax: `void free(void*)`

Task: `free(ptr)` "de-allocate" memory allocated by `ptr=malloc(...)`

Example:

```
List delete(List l, int value) {
    if ( l == NULL ) return NULL;
    l->next = delete(l->next,value);
    if (l->elem == value) {
        List rest;
        rest = l->next;
        free(l);
        return rest;
    }
    else return l;
}
```

Sparse Matrices

In scientific computing, matrices are often sparse.
We'll try to store the matrix lines as linked lists containing pairs of indices/values:

```
struct sparse_elem {
   int index;
   double value;
   struct line *next;
}
```

If we assume that every matrix line has at least one non-zero element, we can store a sparse matrix as an array of (the first) `sparse_elem`'s:

```
typedef sparse_elem* sparse_mat;
```

A Diagonal Sparse Matrix

Example: How to construct a diagonal sparse matrix

```
sparse_mat diagonal(int n)
   int i;
   sparse_mat d;
   d = (sparse_mat) malloc(n*sizeof(struct sparse_elem));
   for(i=0;i<n;i++) {
      d[i].index = i+1;
      d[i].value = 1.0;
      d[i].next = NULL;
   }
   return d;
}
```

`diagonal(int n)` returns an $n \times n$ unity matrix.